Automatic Addition of Fault-Tolerance to Real-Time Programs[1]

Borzoo Bonakdarpour Sandeep S. Kulkarni
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA

Abstract

In this paper, we focus on automated addition of fault-tolerance to an existing fault-intolerant *real-time* program. We consider three levels of fault-tolerance, *failsafe*, *nonmasking*, and *masking*, based on the properties satisfied in the presence of faults. Furthermore, for failsafe and masking fault-tolerance, we introduce two cases, *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. We present sound and complete algorithms with polynomial time complexity in the size of region graphs for the case where soft-failsafe, nonmasking, and soft-masking fault-tolerance is added to an existing real-time program. Furthermore, we propose a sound and complete algorithm with polynomial time complexity in the size of region graphs for adding hard-failsafe fault-tolerance, where the synthesized program is required to satisfy at most one bounded response property in the presence of faults. Moreover, we show that the problem of adding hard masking fault-tolerance, where the synthesized fault-tolerant program is required to satisfy multiple bounded response properties in the presence of faults, is NP-hard in the size of the region graph. Thus, this work characterizes classes of problems where adding fault-tolerance to real-time programs is expected to be feasible and where the complexity is too high.

Keywords: Fault-tolerance, Real-time, Program synthesis, Program transformation, Formal methods.

1 Introduction

Fault-tolerance and real-time properties are crucial assurance requirements in many computing systems. However, since fault-tolerance and real-time properties often impose conflicting constraints on systems, they are not easy to combine. Meeting real-time properties needs predictability and fault-tolerance requires programs to continue to function even in the presence of unanticipated faults. In other words, while satisfaction of timing constraints requires a priori knowledge of the system's temporal operation, fault-tolerance is built on the principle that faults occur unexpectedly and that faults must be handled through some recovery mechanism.

[1]Email: `{borzoo,sandeep}@cse.msu.edu`
Web: `http://www.cse.msu.edu/~{borzoo,sandeep}`
Tel: +1-517-355-2387, Fax: +1-517-432-1061
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE **Automatic Addition of Fault-Tolerance to Real-Time Programs**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**20**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Automated program synthesis is the problem of designing an algorithmic method to find a program that satisfies a required set of properties. Such automated synthesis is desirable, as it ensures that the synthesized program is correct by construction even if its required set of properties have conflicting constraints such as fault-tolerance and real-time. In the existing specification-based synthesis methods, a change in the specification requires us to synthesize from scratch. Thus, it would be advantageous, if we could reuse the previous efforts made to synthesize fault-intolerant real-time programs and somehow *add* fault-tolerance to them. Moreover, such addition is especially useful if the fault-intolerant real-time program is designed manually, e.g., for ensuring that the original program is efficient.

With this motivation, in this paper, we focus on designing synthesis algorithms that solely *add* fault-tolerance to real-time programs . Such synthesis methods are desirable, as it may not be possible to anticipate all faults that a program may be subject to, at design time. Our goal in this work is to concentrate on algorithms with manageable time and space complexity, i.e., complexity that is comparable to the corresponding complexity of existing model checking techniques for fault-tolerant programs in dense real-time model.

Regarding fault-tolerance, we consider three levels, based on the properties satisfied in the presence of faults. Intuitively, a *failsafe* fault-tolerant program does not violate its safety specification even in the presence of faults, i.e., a *bad thing* does not occur when the program is running in the presence of faults. A *nonmasking* program ensures recovery to its normal behavior after the occurrence of faults. A *masking* fault-tolerant program has both properties, i.e., in the presence of faults, it does not violate its safety specification while ensuring recovery to its normal behavior. Regarding real-time, we propose two cases, *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults (cf. Section 3 for examples).

1.1 Related Work

In real-time computing literature, fault-tolerance has mostly been addressed in the context of scheduling algorithms (e.g., [1–5]). In fault-tolerant real-time scheduling, the objective is to find the optimal schedule of a set of tasks on a set of processors *dynamically*, such that the largest possible set of tasks meet their deadlines. Since time complexity is a critical issue in dynamic scheduling, most of the proposed algorithms are in the form of heuristics designed for specific platforms or architectures and for a special type of faults (e.g., transient, fail-stop, Byzantine, etc.).

The problem of synthesizing *untimed* fault-tolerant programs has been studied in the literature from different perspectives. In [6–10], the authors propose synthesis methods, heuristics, and enhancement algorithms for adding fault-tolerance and multitolerance to existing programs in the high (respectively, low) atomicity model, where processes can (respectively, cannot) read and write all the program variables in one atomic step. In [11], Attie, Arora, and Emerson study the problem of synthesizing fault-tolerant concurrent untimed programs from temporal logic specification expressed in CTL formulas.

Synthesis of real-time systems has mostly been studied in the context of timed automata from a gametheoretic perspective [12–19]. In these papers, the common assumption is that the existing program (called a plant) and/or the given specification is *deterministic*. Moreover, since the authors of the aforementioned work consider highly expressive specifications, the complexity of proposed methods are also very high. For example, algorithms presented in [12–15, 18, 19] are EXPTIME-complete. Moreover, deciding the existence of a solution (called a controller) in [16, 17] is 2EXPTIME-complete.

Online fault *detection* in a given dense-timed automaton is studied by Tripakis in [20]. The author proposes a polynomial space online algorithm to design a diagnoser that detects faults in the behavior of the given timed automaton after they occur. In this modeling, it is assumed that (1) the given system is in synchronous model, and (2) faults and errors are the same thing. Bouyer, Chevalier, and D'Souza [21] address the same problem where the diagnoser is supposed to be realizable as a deterministic timed automaton or an event record automaton.

1.2 Contributions

The point of departure of our work from the above related work is as follows. In this paper we (i) consider a generic fault-tolerance framework for real-time programs independent of platform, architecture, and type of faults; (ii) extend the previous work by Kulkarni and Arora [6] for adding fault-tolerance to untimed programs; (iii) consider a general notion of real-time programs that covers both deterministic and nondeterministic programs in both synchronous and asynchronous models; and (iv) consider different levels of fault-tolerance for real-time systems based on satisfaction of properties and timing constraints. Furthermore, we present a class of specifications where we can express typical requirements for specifying real-time and fault-tolerant computing systems and we show that the complexity of synthesis algorithms for this class of specifications is manageable in the sense that they are comparable to existing model checking techniques for real-time programs [22]. The main results in this paper are as follows:

1. We propose a generic formal framework that defines the notions of faults and levels fault-tolerance in the context of real-time programs.
2. We present polynomial time (in the size of the region graphs) sound and complete algorithms that transform fault-intolerant real-time programs into
   (a) soft-failsafe, nonmasking, and soft-masking fault-tolerant programs, and
   (b) hard-failsafe fault-tolerant programs, where the synthesized fault-tolerant program is required to satisfy at most one bounded response property in the presence of faults.
3. We present a sound polynomial time algorithm that transforms a fault-intolerant real-time program into a hard masking fault-tolerant program, where the synthesized fault-tolerant program is required to satisfy at most one bounded response property in the presence of faults.
4. We note that the problem of adding hard masking fault-tolerance, where the synthesized program is required to satisfy multiple bounded response properties in the presence of faults, is NP-hard.

Organization of the paper. In Section 2, we present the preliminary concepts. We formally define the notions of faults and fault-tolerance in the context of real-time programs in Section 3. In Section 4, we formally state the problem of adding fault-tolerance to real-time programs. We present our transformation algorithms and NP-hardness result in Section 5. Then, in Section 6, we answer the potential questions raised about our approach. Finally, in Section 7, we make the concluding remarks and discuss future work.

2 Preliminaries

In this section, we present the preliminary concepts and formal definitions of real-time programs, specifications, and region graphs. Programs are specified in terms of their state space and their transitions [23]. The definition of specifications is adapted from Henzinger [24]. Finally, the notion of region graph is due to Alur and Dill [25].

2.1 Program

A program includes a finite set V of *discrete variables* and a finite set X of *clock variables*. Each discrete variable is associated with a finite *domain D* of values. A *location* is a function that maps each discrete variable to a value from its respective domain. For the set X of clock variables , the set $\Phi(X)$ of *clock constraints* φ is inductively defined by the grammar: $\varphi ::= x \leq c \mid x \geq c \mid x < c \mid x > c \mid \varphi \wedge \varphi$, where $x \in X$ and $c \in \mathbb{Z}_{\geq 0}$. A *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. Furthermore, for $\tau \in \mathbb{R}_{\geq 0}$, $\nu + \tau = \nu(x) + \tau$ for every clock x. Also, for $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation for X which assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clock variables in X.

A *state* of a program (denoted σ) is a pair (s, ν), such that s is a location and ν is a clock valuation for X at location s. Since the domain of clock variables ranges over the real numbers, the *state space* of a program (the set of all possible states) is infinite. A *transition* of a program (denoted (σ_0, σ_1)) is of the form $(s_0, \nu_0) \rightarrow (s_1, \nu_1)$. Transitions are classified into two types:

- **Delay (elapse of time):** for a state $\sigma = (s, \nu)$ and a time *duration* $\delta \in \mathbb{R}_{\geq 0}$ (denoted (σ, δ)), $(s, \nu) \rightarrow (s, \nu + \delta)$.
- **Jump (location switch):** for a state (s_0, ν), a location s_1, and a set λ of clock variables, $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$.

We say a state σ_1 is *passed* by the delay (σ_0, δ) if $\sigma_1 = \sigma_0 + \epsilon$ for some $\epsilon \in \mathbb{R}_{\geq 0}$ such that $\epsilon < \delta$.

A *program* $\mathcal{P}$ is a tuple $\langle S_p, \psi_p \rangle$, where S_p is the state space, and ψ_p is a set of transitions. Let ψ_p^s and ψ_p^d denote the set of jump and delay transitions in ψ_p, respectively. A *state predicate* is a Boolean expression over the variables of $\mathcal{P}$. Note that, in such an expression, a clock constraint must be picked from $\Phi(X)$, i.e., clock variables can only be compared with nonnegative integers. A state predicate can also be expressed as a subset of S_p such that it is definable by the above syntax of clock constraints. A state predicate S is *closed* in the program $\mathcal{P}$ iff $((\forall(\sigma_0, \sigma_1) \in \psi_p^s : (\sigma_0 \in S \Rightarrow \sigma_1 \in S)) \wedge (\forall(\sigma, \delta) \in \psi_p^d : (\sigma \in S \Rightarrow \forall \epsilon \leq \delta : \sigma + \epsilon \in S)))$, i.e., if a jump transition originates in S then it must terminate in S, and if a delay transition originates in a state in S then any state passed by the delay plus the target state must be in S. A timed state sequence $\langle(\sigma_0,\tau_0),(\sigma_1,\tau_1)\cdots\rangle$, where $\tau_i \in \mathbb{R}_{\geq 0}$, is a *computation* of $\mathcal{P}$ iff the following conditions are satisfied: (1) $\forall j > 0 : (\sigma_{j-1},\sigma_j) \in \psi_p$, (2) if it is finite and terminates in (σ_l,τ_l) then there does not exist state σ such that $(\sigma_l,\sigma) \in \psi_p$, and (3) the sequence $\langle\tau_0,\tau_1\cdots\rangle$ satisfies the following constraints:

Monotonicity: $\tau_i \leq \tau_{i+1}$ for all $i \in \mathbb{N}$.

Divergence: For all $t \in \mathbb{R}_{\geq 0}$, there exists j such that $\tau_j \geq t$.

The *projection* of a set of program transitions ψ_p on state predicate S (denoted $\psi_p|S$) is the set of transitions $\{(\sigma_0,\sigma_1) \mid (\sigma_0,\sigma_1) \in \psi_p^s \wedge \sigma_0,\sigma_1 \in S\} \cup \{(\sigma,\delta) \mid (\sigma,\delta) \in \psi_p^d \wedge \sigma \in S \wedge (\forall\epsilon \leq \delta : \sigma+\epsilon \in S)\}$. I.e., $\psi_p|S$ consists of jump transitions of ψ_p that start in S and end in S, and delay transitions of ψ_p that start and remain in S continuously.

2.2 Specification

A *specification* (or *property*), denoted Σ, is a set of timed state sequences of the form $\langle(\sigma_0,\tau_0),(\sigma_1,\tau_1)\cdots\rangle$. Following Henzinger [24], we require the sequence $\langle\tau_0,\tau_1\cdots\rangle$ to satisfy monotonicity and divergence. We now define what it means for a program $\mathcal{P}$ to satisfy a specification Σ. Given a program $\mathcal{P}$, a state predicate S, and a specification Σ, we write $\mathcal{P} \models_S \Sigma$ and say that program $\mathcal{P}$ *satisfies* Σ *from* S iff (1) S is closed in $\mathcal{P}$, and (2) every computation of $\mathcal{P}$ that starts where S is true is in Σ. If $\mathcal{P} \models_S \Sigma$ and $S \neq \{\}$, we say S is an *invariant* of $\mathcal{P}$ for Σ.

Notation. Whenever the specification is clear from the context, we will omit it; thus, "S is an invariant of $\mathcal{P}$" abbreviates "S is an invariant of $\mathcal{P}$ for Σ".

We say that program $\mathcal{P}$ *maintains* Σ iff for all finite timed state sequences α of $\mathcal{P}$, there exists a timed state sequence β such that $\alpha\beta \in \Sigma$. Similarly, we say that $\mathcal{P}$ *violates* Σ iff it is not the case that $\mathcal{P}$ maintains Σ. Note that, the definition of *maintains* identifies the property of finite timed state sequences, whereas the definition of *satisfies* expresses the property of infinite timed state sequences.

Following Alpern and Schneider [26] and Henzinger [24], we let the specification consist of a *liveness specification* and a *safety specification*. The liveness specification is represented by a set of infinite computations. A program satisfies the liveness specification, if every computation prefix of the program has a suffix that is in the liveness specification.

Remark 2.1: In the synthesis problem, we begin with an initial fault-intolerant program that satisfies its specification (including the liveness specification) in the absence of faults. In Section 5, we show that our synthesis algorithms preserve liveness specification. Hence, the liveness specification need not be specified explicitly.

Regarding safety, in synthesis algorithms presented in this paper, we let the safety specification consist of (1) a set Σ_{bt} of bad transitions that should not occur in the program computation, i.e., a subset of $\{(\sigma_0,\sigma_1) \mid \sigma_0,\sigma_1 \in S_p\}$, and (2) a conjunction of zero or more *bounded response* properties of the form

$\Sigma_{br} \equiv ((P_1 \mapsto_{\leq \delta_1} Q_1) \ \wedge \ (P_2 \mapsto_{\leq \delta_2} Q_2) \ \wedge \ ... \wedge (P_m \mapsto_{\leq \delta_m} Q_m))$, i.e., it is always the case that a state in P_i is followed by a state in Q_i within δ_i time units, where P_i and Q_i are state predicates and $\delta_i \in \mathbb{Z}_{\geq 0}$, for all i such that $1 \leq i \leq m$. Observe that it is possible to trivially translate this concise representation of safety into the corresponding set of infinite computations. The same concept is applicable to definitions of *maintains* and *violates*.

2.3 Region Graph

Given a program $\mathcal{P}\langle S_p, \psi_p\rangle$, in order to reason about properties of $\mathcal{P}$, one must deal with the infinite state space S_p. Alur and Dill [25] propose construction of a finite quotient as a solution for dealing with the infinite state space. This construction uses an equivalence relation, called *region equivalence* (denoted $\cong$), on the state space that equates two states with the same location, is defined over the set of all clock valuations for X. For two clock valuations ν and μ, $\nu \cong \mu$ iff:

1. $\forall x \in X : ((\lfloor \nu(x) \rfloor = \lfloor \mu(x) \rfloor) \ \vee \ (\nu(x), \mu(x) > c_x))$,
2. $\forall x, y \in X \ : \ ((\nu(x) < c_x \ \wedge \ \nu(y) < c_y)) : (\langle \nu(x) \rangle \ < \ \langle \nu(y) \rangle$ iff $\langle \mu(x) \rangle \ < \ \langle \mu(y) \rangle)$, and
3. $\forall x \in X : \nu(x) \ < \ c_x : (\langle \nu(x) \rangle = 0$ iff $\langle \mu(x) \rangle = 0)$,

where c_x is the largest integer c, such that x is compared with c in a clock constraint, $\langle \tau \rangle$ denotes the fractional part, and $\lfloor \tau \rfloor$ denotes the integral part of τ and for any $\tau \in \mathbb{R}_{\geq 0}$. A *clock region* for $\mathcal{P}$ is an equivalence class of clock valuations induced by $\cong$. Note that, there are only finite number of clock regions.

A *region* is a pair (s, ρ), where s is a location and ρ is a clock region. Using the region equivalence relation, we construct the *region graph* of $\mathcal{P}\langle S_p, \psi_p\rangle$ (denoted $R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$) as follows. Vertices of $R(\mathcal{P})$ (denoted S_p^r) are regions. Edges of $R(\mathcal{P})$ (denoted ψ_p^r) are of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$ iff for some clock valuations $\nu_0 \in \rho_0$ and $\nu_1 \in \rho_1$, $(s_0, \nu_0) \rightarrow (s_1, \nu_1)$ is a transitions in ψ_p. We say that a region (s_0, ρ_0) of region graph $R(\mathcal{P})$ is a *deadlock region* iff for all regions (s_1, ρ_1), there does not exist an edge of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$.

A *region predicate* S^r with respect to a state predicate S is defined by $S^r = \{(s, \rho) \mid \exists (s, \nu) : ((s, \nu) \in S \ \wedge \ \nu \in \rho)\}$. Likewise, the region predicate with respect to invariant S of a program $\mathcal{P}$ is called *region invariant* S^r. The projection of a set of edges ψ_p^r on region predicate S^r (denoted $\psi_p^r | S^r$) is the set of edges $\{(r_0, r_1) \mid (r_0, r_1) \in \psi_p^r \ \wedge \ r_0, r_1 \in S^r\}$.

Based on the above description to construct a region graph, in our synthesis algorithms in Section 5, we transform a real-time program $\mathcal{P}\langle S_p, \psi_p\rangle$ into its corresponding region graph $R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$ by invoking the subroutine ConstructRegionGraph. We also let this subroutine take state predicates and sets of transitions in $\mathcal{P}$ (e.g., S and Σ_{bt}) and return the corresponding regions predicates and sets of edges in $R(\mathcal{P})$ (e.g., S^r and Σ_{bt}^r).

A clock region β is a *time-successor* of a clock region α iff for each $\nu \in \alpha$, there exists $\tau \in \mathbb{R}_{\geq 0}$, such that $\nu + \tau \in \beta$, and $\nu + \tau' \in \alpha \cup \beta$ for all $\tau' < \tau$. We call a region (s, ρ) a *boundary region*, if for each $\nu \in \rho$

and for any $\tau \in \mathbb{R}_{\geq 0}$, ν and $\nu + \tau$ are not equivalent. A region is *open*, if it is not a boundary region. A region (s, ρ) is called an *end region*, if for all $\nu \in \rho$ and for all clocks x, $\nu(x) > c_x$.

3 Faults and Fault-Tolerance in Real-Time Programs

In this section, we extend formal definitions of faults and fault-tolerance due to Arora and Gouda [27] and Arora and Kulkarni [28], so that they fit in the context of real-time programs.

The faults that a program is subject to are systematically represented by transitions. A class of *faults* f for program $\mathcal{P}\langle S_p, \psi_p\rangle$ is a subset of the set $S_p \times S_p$. We use $\psi_p[]f$ to denote the transitions obtained by taking the union of the transitions in ψ_p and the transitions in f.

We say that a state predicate T is an f-span (read as *fault-span*) of $\mathcal{P}$ from S iff the following conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $\psi_p[]f$. Observe that for all computations of $\mathcal{P}$ that start at states where S is true, T is a boundary in the state space of $\mathcal{P}$ up to which (but not beyond which) the state of $\mathcal{P}$ may be perturbed by the occurrence of the transitions in f. Similar to the notion of region invariant (cf. Subsection 2.3), the region predicate with respect to fault-span T of a program $\mathcal{P}$ is called *region fault-span* T^r. Likewise, f^r denotes the set of faults edges in $R(\mathcal{P})$ that correspond to fault transitions f in $\mathcal{P}$.

As we defined the computations of $\mathcal{P}$, we say that a timed state sequence, $\langle(\sigma_0, \tau_0), (\sigma_1, \tau_1), \cdots\rangle$, is a *computation of* $\mathcal{P}$ *in the presence of* f iff the following four conditions are satisfied: (1) $\forall j > 0 : (\sigma_{j-1}, \sigma_j) \in (\psi_p \cup f)$, (2) if $\langle\sigma_0, \sigma_1, \cdots\rangle$ is finite and terminates in state (σ_l, τ_l) then there does not exist state σ such that $(\sigma_l, \sigma) \in \psi_p$, (3) $\langle\tau_0, \tau_1 \cdots\rangle$ satisfies monotonicity and divergence, and (4) $\exists n \geq 0 : (\forall j > n : (\sigma_{j-1}, \sigma_j) \in \psi_p)$.

In this paper, we consider three levels of fault-tolerance, failsafe, nonmasking, and masking, based on the properties satisfied in the presence of faults. For failsafe and masking fault-tolerance, we propose two cases, *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. To motivate the idea of soft and hard fault-tolerance let us consider the railroad crossing problem. Suppose that a train is approaching a railroad crossing. The safety specification requires "*if the train is crossing, the gate should be closed*". Also, the bounded response property requires that "*once the gate is closed, it should reopen within 5 minutes*". In this example, it may be catastrophic if the train is crossing while the gate is open due to occurrence of faults. On the other hand, if the gate remains closed for more than 5 minutes due to occurrence of faults, the outcome is not disastrous. Thus, depending upon the outcome of violation of a safety specification, the desired fault-tolerance requirement changes. Hence, in the railroad crossing problem the desired requirement is the system must tolerate faults that cause the gate to be open while the train is crossing and, hence, this system must be *soft fault-tolerant*. Intuitively, a soft fault-tolerant real-time program is not required to satisfy its timing constraints in the presence of faults.

Now, consider a system that controls internal pressure of a boiler. Suppose that in this system, the safety specification requires that once a pressure gauge reads 30 pounds per square inch, the controller must issue a command to open a valve within 20 seconds. In such a system, if occurrence of faults causes the controller not to respond within the required time, the outcome may be disastrous. Thus, our boiler controller must satisfy its timing constraints even in the presence of faults. In other words, the boiler controller must be *hard fault-tolerant*. Intuitively, a hard fault-tolerant real-time program must satisfy its timing constraints in the presence of faults.

We now present formal definitions of different levels of fault-tolerance. Let specification Σ consist of Σ_{bt} and Σ_{br}. We say that $\mathcal{P}$ is *soft-failsafe f-tolerant from S for Σ* iff the following conditions hold: (1) $\mathcal{P} \models_S \Sigma_{bt}$, (2) $\mathcal{P} \models_S \Sigma_{br}$, and (3) there exists T such that T is an f-span of $\mathcal{P}$ from S, and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{bt} from T. A program $\mathcal{P}$ is *hard-failsafe f-tolerant from S for Σ* iff $\mathcal{P}$ is soft-failsafe f-tolerant from S for Σ and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{br} from T.

Since a nonmasking fault-tolerant program need not satisfy safety in the presence of faults, $\mathcal{P}$ is *nonmasking f-tolerant from S for Σ with recovery time δ*, where $\delta \in \mathbb{Z}_{\geq 0}$, iff the following conditions hold: (1) $P \models_S \Sigma_{bt}$, (2) $\mathcal{P} \models_S \Sigma_{br}$, and (3) there exists T such that T is an f-span of $\mathcal{P}$ from S, and every computation of $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ that starts from a state in T, reaches a state in S within δ time units.

A program $\mathcal{P}\langle S_p, \psi_p\rangle$ is *soft-masking f-tolerant from S for Σ with recovery time δ*, where $\delta \in \mathbb{Z}_{\geq 0}$, iff the following conditions hold: (1) $\mathcal{P} \models_S \Sigma_{bt}$, (2) $\mathcal{P} \models_S \Sigma_{br}$, (3) there exists T such that T is an f-span of $\mathcal{P}$ from S and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{bt} from T, and (4) every computation of $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ that starts from a state in T, reaches a state in S within δ time units. A program $\mathcal{P}\langle S_p, \psi_p\rangle$ is *hard-masking f-tolerant from S for Σ with recovery time δ*, where $\delta \in \mathbb{Z}_{\geq 0}$, iff $\mathcal{P}$ is soft-masking f-tolerant from S for Σ with recovery time δ, and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{br} from T.

Notation. Whenever the specification Σ and the invariant S are clear from the context, we omit them; thus, "f-tolerant" abbreviates "f-tolerant from S for Σ".

Assumption 3.1: Since the program $\mathcal{P}$ satisfies $\Sigma_{br} \equiv ((P_1 \mapsto_{\leq \delta_1} Q_1) \wedge (P_2 \mapsto_{\leq \delta_2} Q_2) \wedge ... \wedge (P_m \mapsto_{\leq \delta_m} Q_m))$ in the absence of faults (cf. Remark 2.1), without loss of generality, we assume that for each bounded response property $(P_i \mapsto_{\leq \delta_i} Q_i)$, where $1 \leq i \leq m$, the intolerant program already has a clock variable that is reset on transitions that go from a state in $\neg P_i$ to a state in P_i. This assumption simplifies dealing with the given bounded response property, as we are ensured that the program itself keeps track of time when P_i becomes true. In case such a clock does not exist, we can simply add it without changing semantics of the given program.

Assumption 3.2: We assume that faults are immediately detectable and that given a state of the program, we can determine the number of faults that have occurred in reaching that state. (For example, one can achieve this if the program has a variable that stores how many faults have occurred in a program computation.) This assumption is needed only for hard fault-tolerance.

Assumption 3.3: We assume that the number of occurrence of faults in a program computation is bounded by a pre-specified value n. This assumption is required since for commonly considered faults, it can be shown that *bounded-time recovery* in the presence of unbounded occurrence of faults is impossible.

4 Problem Statement

Given are a fault-intolerant real-time program $\mathcal{P}\langle S_p, \psi_p\rangle$, its invariant S, a set of faults f, and a safety specification Σ such that $\mathcal{P} \models_S \Sigma$. Our goal is to synthesize a real-time program $\mathcal{P}'\langle S_p, \psi'_p\rangle$ with invariant S' such that $\mathcal{P}'$ is f-tolerant from S' for Σ.

As mentioned in the introduction, our synthesis method obtains $\mathcal{P}'$ from $\mathcal{P}$ by adding fault-tolerance *alone* to $\mathcal{P}$, i.e., $\mathcal{P}'$ does not introduce new behaviors to $\mathcal{P}$ when no faults have occurred. We now describe how we formulate the problem. Observe that:

1. If S' contains states that are not in S then, in the absence of faults, $\mathcal{P}'$ may include computations that start outside S. Since $\mathcal{P}' \models_{S'} \Sigma$, it would imply that $\mathcal{P}'$ is using a new way to satisfy Σ in the absence of faults (since $\mathcal{P}$ satisfies Σ only from S). Therefore, we require that $S' \subseteq S$.
2. If $\psi'_p|S'$ contains a transition that is not in $\psi_p|S'$ then $\mathcal{P}'$ can use this transition in order to satisfy Σ in the absence of faults. Since this was not permitted in $\mathcal{P}$, we require that $\psi'_p|S' \subseteq \psi_p|S'$.

Thus, the synthesis problem is as follows (This definition will be instantiated for (soft and hard) failsafe, nonmasking, and (soft and hard) masking f-tolerance):

Problem Statement 4.1. Given $\mathcal{P}\langle S_p, \psi_p\rangle$, S, Σ, and f such that $\mathcal{P} \models_S \Sigma$.
Identify $\mathcal{P}'\langle S_p, \psi'_p\rangle$ and S' such that

$(C1)$ $S' \subseteq S$

$(C2)$ $\psi'_p|S' \subseteq \psi_p|S'$, and

$(C3)$ $\mathcal{P}'$ is f-tolerant from S' for Σ. $\square$

5 Adding Fault-Tolerance to Real-Time Programs

In this section, we present our synthesis algorithms and NP-hardness result for adding fault-tolerance to an existing real-time program. In particular, in Subsection 5.1, we describe our algorithms for adding (soft and hard) failsafe fault-tolerance. In Subsection 5.2, we describe how we add nonmasking fault-tolerance. In Subsection 5.3, we describe automated addition of (soft and hard) masking fault-tolerance. Finally, in Subsection 5.4, we consider the case of hard masking fault-tolerance where two or more timing constraints must be met in the presence of faults.

5.1 Automated Addition of Failsafe Fault-Tolerance to Real-Time Programs

In this subsection, we first present our algorithm for adding soft-failsafe fault-tolerance. Then, we describe our algorithm for adding hard-failsafe, where the synthesized program is required to satisfy at most one bounded response property in the presence of faults. As mentioned in Subsection 2.2, the safety specification identifies a set Σ_{bt} of bad transitions that should not occur in any program computation, and a conjunction Σ_{br} of multiple bounded response properties. Also, recall that in the presence of faults, a soft-failsafe program is required to maintain only Σ_{bt}, whereas a hard-failsafe program should maintain both Σ_{bt} and Σ_{br}.

5.1.1 Adding Soft-Failsafe Fault-Tolerance

In order to synthesize a soft-failsafe program, we should generate a program $\mathcal{P}'$, such that a bad transition $(\sigma_0, \sigma_1) \in \Sigma_{bt}$ does not occur in any computation of $\mathcal{P}'$ in the presence of faults. Towards this end, we adapt the proposed algorithm in [6] that adds failsafe fault-tolerance to *untimed* programs.

We now describe our algorithm Add_SoftFalisafe (cf. Figure 1) in detail. First, we transform the real-time program $\mathcal{P}\langle S_p, \psi_p\rangle$, invariant S, a set f of fault transitions, and a set Σ_{bt} of bad transitions into a region graph $R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$, region invariant S^r, fault edges f^r, and bad edges Σ_{bt}^r (Line A1). This is achieved by invoking the subroutine ConstructRegionGraph, as described in Subsection 2.3. Then, the algorithm adds failsafe fault-tolerance to $R(\mathcal{P})$, so that no edge of Σ_{bt}^r occurs in computations of $R(\mathcal{P})$. This is achieved by invoking the subroutine Add_UntimedFailsafe (Line A2).

The subroutine Add_UntimedFailsafe (cf. Figure 1) first finds the set ms of regions and the set mt of edges from where safety of $\mathcal{P}$ may be violated by faults alone (lines C1, C2). Next, it removes such regions (respectively, edges) from the region invariant S^r (respectively, set of edges ψ_p^r) of $R(\mathcal{P})$. This removal may create deadlock regions. Hence, next, the subroutine removes deadlock regions from S^r (Line C3), ensures closure of S^r in ψ_p^r (Line C5), and returns a failsafe region graph $R(\mathcal{P}')\langle S_p^r, \psi_p'^r\rangle$ (Line C6).

Finally, The algorithm Add_SoftFalisafe transforms the region graph $R(\mathcal{P}')$ back into a real-time program $\mathcal{P}'$ (Line A3).

Theorem 5.1. The algorithm Add_SoftFalisafe is sound and complete.

For reasons of space, we refer the reader to [29] for proofs of all theorems in this paper.. □

Theorem 5.2. The problem of adding soft-failsafe fault-tolerance to a real-time program is in PSPACE. □

5.1.2 Adding Hard-Failsafe Fault-Tolerance with a Single Bounded Response Property

In this subsection, we consider the case that a hard-failsafe fault-tolerant program is required to satisfy at most one bounded response property in the presence of faults. In other words, $\Sigma_{br} \equiv P \mapsto_{\leq\delta} Q$. Towards this end, we need to generate a program $\mathcal{P}'$, such that it maintains both Σ_{bt} and Σ_{br} in the presence of faults. In other words, a bad transition $(\sigma_0, \sigma_1) \in \Sigma_{bt}$ occurs in no computation of $\mathcal{P}'$. Moreover, if a computation of $\mathcal{P}'$ reaches a state in P then it reaches a state in Q within δ units of time even in the presence of faults. To this end, we first add soft-failsafe fault-tolerance to $R(\mathcal{P})$ to ensure that $\mathcal{P}'$ maintains Σ_{bt} in the presence of faults. Then, we transform $R(\mathcal{P})$ to an ordinary weighted directed graph (called MaxDelay digraph). To ensure that the maximum delay to reach a state in Q from each state in P is at most δ time units in the presence of faults, we extract a subgraph of the MaxDelay digraph, such that the longest distance between the vertices that correspond to the states in P and Q is at most δ. Before we present our algorithm for adding hard-failsafe fault-tolerance in detail, we reiterate how to construct a MaxDelay digraph from [30].

Construction of MaxDelay digraph. We now describe the subroutine ConstructMaxDelayGraph that transforms a region graph to a MaxDelay digraph. The subroutine takes a region graph $R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$ and a

```
Add_SoftFailsafe($\mathcal{P}\langle S_p, \psi_p\rangle$ :real-time program $f$ :transitions, $S$: state predicate, $\Sigma_{bt}$: specification)
{
    $R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$, $S^r$, $f^r$, $\Sigma_{bt}^r$ := ConstructRegionGraph($\mathcal{P}\langle S_p, \psi_p\rangle$, $S$, $f$, $\Sigma_{bt}$);    (A1)
    $\psi_p'^r, S'^r, mt$ := Add_UntimedFailsafe($R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, f^r, S^r, \Sigma_{bt}^r$);    (A2)
    $\mathcal{P}'\langle S_p, \psi_p'\rangle, S'$ := ConstructRealTimeProgram($R(\mathcal{P})\langle S_p^r, \psi_p'^r\rangle, S'^r$)    (A3)
}

Add_HardFailsafe($\mathcal{P}\langle S_p, \psi_p\rangle$ :real-time program $f$ :transitions, $S, P, Q$: state predicate, $\Sigma_{bt}$: specification, $n, \delta$: **integer**)
{
    $R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, S^r, P^r, Q^r, f^r, \Sigma_{bt}^r$ := ConstructRegionGraph($\mathcal{P}\langle S_p, \psi_p\rangle, S, P, Q, f, \Sigma_{bt}$);    (B1)
    $\psi_p^r, S^r, mt$ := Add_UntimedFailsafe($R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, f^r, S^r, \Sigma_{bt}^r$);    (B2)

    **repeat**
        *IsQRemoved* := *false*;    (B3)
        $\psi_p^r := \psi_p^r \cup \{((s_0,\rho_0),(s_1,\rho_1)) \mid (s_0,\rho_0) \notin S^r \wedge$
                $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0$ : $(\exists \lambda \subseteq X : \rho_1 = \rho_2[\lambda := 0])\}$;    (B4)
        $\psi_p^r, ns$ := Add_BoundedRecovery($R(\mathcal{P})\langle S_p^r - ms, \psi_p^r - mt\rangle, f^r, P^r, Q^r, n, \delta$);    (B5)
        $rs := \{r_0 \mid \exists r_1, r_2, ...r_n : (\forall j : 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \wedge r_n \in ns \wedge r_n \in P^r\}$;    (B6)
        $rt := \{(r_0, r_1) \mid (r_0, r_1) \in \psi_{p_1}^r \wedge r_1 \in rs)\}$;    (B7)
        $S'^r$ := RemoveDeadlocks($S^r - (ns \cup rs), Q^r, \psi_p^r - rt$);    (B8)
        **if** ($S'^r = \{\}$) **then** declare no hard-failsafe $f$-tolerant program $\mathcal{P}'$ exists; **exit**;    (B9)
        **if** ($\exists r_0 \in Q^r : (r_0 \in S^r \wedge r_0 \notin S'^r)$) **then**    (B10)
            *IsQRemoved* := *true*;    (B11)
            $S^r := S'^r$;    (B12)
            $Q^r := Q^r - \{r_0\}$;    (B13)
            $\psi_p^r := \psi_p^r - \{(r, r_0), (r_0, r) \mid r \in S^r\}$;    (B14)
    **until** (*IsQRemoved* = *false*)

    $\psi_p'^r$ := EnsureClosure($\psi_p'^r, S'^r$);    (B15)
    $\mathcal{P}'\langle S_p, \psi_p'\rangle, S'$ := ConstructRealTimeProgram($R(\mathcal{P})\langle S_p^r, \psi_p'^r\rangle, S'^r$)    (B16)
}

Add_UntimedFailsafe($R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$: region graph, $f^r$ : set of edges, $S^r$ : region predicate, $\Sigma_{bt}^r$ : specification)
{
    $ms := \{r_0 \mid \exists r_1, r_2, ...r_n : (\forall j \mid 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \wedge (r_{n-1}, r_n) \in \Sigma_{bt}^r\}$;    (C1)
    $mt := \{(r_0, r_1) \mid (r_1 \in ms) \vee ((r_0, r_1) \in \Sigma_{bt}^r)\}$;    (C2)
    $S^r$ := RemoveDeadlocks($S^r - ms, \{\}, \psi_p^r - mt$);    (C3)
    **if** ($S^r = \{\}$) **then** declare no soft/hard-failsafe $f$-tolerant program $\mathcal{P}'$ exists;**exit**;    (C4)
    $\psi_p^r$ :=EnsureClosure($\psi_p^r - mt, S^r$);    (C5)
    **return** $\psi_p^r, S^r, mt$    (C6)
}

RemoveDeadlocks($S^r, Q^r$ : region predicate, $\psi_p^r$ : set of edges)
// *Returns the largest subset of $S^r$ from where all computations of $R(\mathcal{P})$ are infinite*
{
    **while** ($\exists r_0 \mid r_0 \in S^r : (\forall r_1 \in S^r : (r_0, r_1) \notin \psi_p^r)$)
        $S^r := S^r - \{r_0\}$;
        **if** ($r_0 \in Q^r$) **then break**;
    **return** $S^r$
}

EnsureClosure($\psi_p^r$ : set of edges, $S^r$ : region predicate)
    { **return** $\psi_p^r - \{(r_0, r_1) \mid r_0 \in S^r \wedge r_1 \notin S^r\}$}
```

Figure 1: Addition of Failsafe Fault-Tolerance

set f^r of fault edges as input, and constructs a MaxDelay digraph $G\langle V, A\rangle$ as follows. Vertices of G consists of the regions in $R(\mathcal{P})$.

Notation: We denote the weight of an arc (v_0, v_1) by $\mathit{Weight}(v_0, v_1)$. Let γ denote a bijective function that maps each region $r \in S_p^r$ to its corresponding vertex in G; i.e., $\gamma(r)$ is a vertex of G that represents region r of $R(\mathcal{P})$. Also, let γ^{-1} denote the inverse of γ; i.e., $\gamma^{-1}(v)$ is the region of $R(\mathcal{P})$ that corresponds to vertex v in V. Let Γ be a function that maps a region predicate in $R(\mathcal{P})$ to the corresponding set of vertices of G and let Γ^{-1} be its inverse. Finally, for a boundary region r with respect to clock variable x, we denote the value of x by $r.x$ (equal to some constant in $\mathbb{Z}_{\geq 0}$).

Arcs of G consists of the following:

- Arcs of weight 0 from v_0 to v_1, if $\gamma^{-1}(v_0) \rightarrow \gamma^{-1}(v_1)$ represents a jump transition in $R(\mathcal{P})$.
- Arcs of weight $c' - c$, where $c, c' \in \mathbb{Z}_{\geq 0}$ and $c' > c$, from v_0 to v_1, if $\gamma^{-1}(v_0)$ and $\gamma^{-1}(v_1)$ are both boundary regions with respect to clock variable x_i, such that $\gamma^{-1}(v_0).x_i = c$, $\gamma^{-1}(v_1).x_i = c'$, and there is a path in $R(\mathcal{P})$ from $\gamma^{-1}(v_0)$ to $\gamma^{-1}(v_1)$, which does not reset x_i.
- Arcs of weight $c' - c - \epsilon$, where $c, c' \in \mathbb{Z}_{\geq 0}$, $c' > c$, and $\epsilon \ll 1$, from v_0 to v_1 , if (1) $\gamma^{-1}(v_0)$ is a boundary region with respect to clock x_i, (2) $\gamma^{-1}(v_1)$ is an open region whose time-successor $\gamma^{-1}(v_2)$ is a boundary region with respect to clock x_i, (3) $\gamma^{-1}(v_0) \rightarrow \gamma^{-1}(v_1)$ represents a delay transition in $R(\mathcal{P})$, and (4) $\gamma^{-1}(v_0).x_i = c$ and $\gamma^{-1}(v_2).x_i = c'$.
- Self-loop arcs of weight ∞ at vertex v, if $\gamma^{-1}(v)$ is an end region.

In order to compute the maximum delay between regions in P^r and Q^r, it suffices to find the longest distance between $\Gamma(P^r)$ and $\Gamma(Q^r)$ in G. Note that, strongly connected components reachable from $\Gamma(P^r)$ containing an arc of nonzero weight cause maximum delay of infinity.

We now describe our algorithm Add_HardFailsafe (cf. Figure 1) in detail. The algorithm takes a real-time program $\mathcal{P}$ with invariant S, a set of fault transitions f, a set of bad transitions Σ_{bt}, a bounded response property $\Sigma_{br} \equiv P \mapsto_{\leq \delta} Q$, the maximum number of occurrence of faults n (cf. Assumption 3.3), and returns a hard-failsafe program $\mathcal{P}'\langle S_p, \psi_p'\rangle$ with invariant S'. First, we transform $\mathcal{P}$ into its region graph $R(\mathcal{P})$ (Line B1). Let P^r and Q^r be region predicates with respect to state predicates P and Q, respectively. Then, to ensure that $\mathcal{P}'$ maintains Σ_{bt}, we add soft-failsafe fault-tolerance to $R(\mathcal{P})$ (Line B2). Next, we modify $R(\mathcal{P})$, such that any computation that starts from a region in P^r, reaches a region in Q^r in at most δ time units even in the presence of faults. Towards this end, we compute the set of regions and edges from where Σ_{br} is maintained (lines B3-B14). In particular, to ensure that Q is reachable from the states in $P \wedge \neg S$, we add edges that start from each region in $S_p^r - S^r$ and go to regions where the time monotonicity condition is preserved (Line B4). Now, we invoke the subroutine Add_BoundedRecovery to ensure that $P \mapsto_{\leq \delta} Q$ is maintained in the presence of faults (Line B5).

```
Add_BoundedRecovery($R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$: region graph, $f^r$: set of edges, $P^r, Q^r$: region predicate, $n, \delta$: integer)
// Adds bounded-time recovery from $P^r$ to $Q^r$ in the presence of $f^r$
{
    $G\langle V, A\rangle :=$ ConstructMaxDelayGraph($R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, f^r$);                                   (D1)
    // Let $G^i\langle V^i, A^i\rangle$ be the portion of G, in which $(n-i)$ faults have occurred, where $0 \leq i \leq n$

    for each vertex $v \in V^0$ : $Rank(v) :=$ Length of the shortest path from $v$ to a vertex in $\Gamma(Q^r)^0$;        (D2)
    for $i = 1$ to $n$                                                                                   (D3)
        for each vertex $v_0 \in V^i$ :                                                                  (D4)
            $V_f := \{v_1 \mid (v_1 \in V^{i-1} \wedge (\gamma^{-1}(v_0), \gamma^{-1}(v_1)) \in f^r)\}$;                          (D5)
            if $V_f \neq \{\}$ then                                                                      (D6)
                $MinRank(v_0) := \max\{(Rank(v_1) + Weight(v_0, v_1))$ for all $v_1 \in V_f\}$;                  (D7)
            else $MinRank(v_0) := 0$;                                                                    (D8)
        AdjustShortestPaths($G^i\langle V^i, A^i\rangle, \Gamma(P^r)^i, \Gamma(Q^r)^i$);                                       (D9)

    // Constructing a subgraph of each portion such that the longest distance between $\Gamma(P^r)$ and $\Gamma(Q^r)$ is at most $\delta$
      and then adding the arcs and vertices that do not appear on paths from $\Gamma(P^r)$ to $\Gamma(Q^r)$
    for $i = 0$ to $n$                                                                                   (D10)
        $G'^i\langle V'^i, A'^i\rangle = \{\}$;                                                                      (D11)
        for each vertex $v \in \Gamma(P^r)^i$ :                                                               (D12)
            if $Rank(v) \leq \delta$ then                                                                (D13)
                $\Pi :=$ the shortest path from $v$ to a vertex in $\Gamma(Q^r)^i$;                                (D14)
                $V'^i := V'^i \cup \{u \mid u$ is on $\Pi\}$;                                                    (D15)
                $A'^i := A'^i \cup \{a \mid a$ is on $\Pi\}$;                                                    (D16)
        $A'^i := A'^i \cup \{(u, v) \mid (u, v) \in A^i \wedge (u \notin V'^i \vee (u \in \Gamma(Q^r)^i))\}$;                     (D17)
        $V'^i := (V'^i \cup \{u \mid (\exists v : (u, v) \in A'^i \vee (v, u) \in A'^i)\})$;                           (D18)

    // Transforming weighted digraph G into a region graph
    $\psi_p'^r := \{(r_0, r_1) \mid (r_0, r_1) \in \psi_p^r \wedge (\gamma(r_0), \gamma(r_1)) \in A'\} \cup$
        $\{(r_1, r_2) \mid (r_1, r_2) \in \psi_p^r \wedge (\gamma(r_1), \gamma(r_2)) \notin A' \wedge \exists r_0 : Weight(\gamma(r_0), \gamma(r_1)) = 1 - \epsilon\}$;   (D19)
    $ns := \{r \mid \gamma(r) \in V - V'\}$;                                                                 (D20)

    return $\psi_p'^r, ns$                                                                              (D21)
}
AdjustShortestPaths($G^i\langle V^i, A^i\rangle$ : directed weighted graph, $V_p, V_q$: set of vertices)
// Adjusts the rank of each vertex based on the ranks computed in Add_BoundedRecovery
{
    for each vertex $v \in V_p$ apply Dijkstra's shortest path with the following change:
        if Dijkstra's shortest path computes a length less than $MinRank(v)$ then
            $Rank(v) := MinRank(v)$;                                                                     (D22)
        else $Rank(v) :=$ length of Dijkstra's shortest path from $v$ to $V_q$                            (D23)
}
```

Figure 2: Addition of Bounded-Time Recovery in the Presence of Faults

The subroutine Add_BoundedRecovery (cf. Figure 2) adds bounded-time recovery to a given region graph as follows. First, it transforms the given region graph $R(\mathcal{P})$ to a MaxDelay digraph $G\langle V, A\rangle$ (Line D1). Recall that, by Assumption 3.2, faults are detectable and $\mathcal{P}$ already has a variable that shows how many faults have occurred in a computation. Thus, let $G^i\langle V^i, A^i\rangle$ be the portion of G, in which $n - i$ faults have occurred, where $0 \leq i \leq n$. More specifically, initially, a computation starts from portion G^n, where no faults have occurred. If a fault occurs in a computation that is currently in portion G^i, the computation will proceed in portion G^{i-1}. Obviously, if n faults occur then the computation proceeds in portion G^0 and no faults will occur

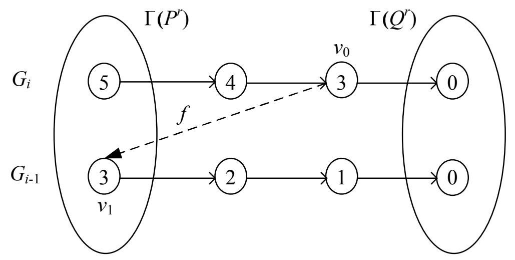

Figure 3: Adjusted Shortest Path.

in that computation. We use these portions to see whether it is possible to reach a vertex in $\Gamma(Q^r)$ from each vertex in $\Gamma(P^r)$ within δ time units.

Next, we rank vertices of all portions of G using a modified Dijkstra's shortest path algorithm, which takes fault perturbations into account (lines D2-D9 and D22-D23). More specifically, since no faults occur in G^0, we first let the rank of all vertices $v \in V^0$ be the length of Dijkstra's shortest path from v to a vertex in $\Gamma(Q^r)^0$ (Line D2). Now, let v_0 be a vertex in V^i, where $1 \leq i \leq n$, and let v_1 be a vertex in V^{i-1}, such that $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ is a fault edge in $R(\mathcal{P})$ and both v_0 and v_1 are on a path from $\Gamma(P^r)$ to $\Gamma(Q^r)$. There exist two cases: (1) the fault edge $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ decreases or does not change the computation delay, i.e, the shortest distance from v_1 to a vertex in $\Gamma(Q^r)^{i-1}$ is less than or equal to the shortest distance from v_0 to a vertex in $\Gamma(Q^r)^i$, and (2) the fault edge $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ increases the computation delay, i.e., the shortest distance from v_1 to a vertex in $\Gamma(Q^r)^{i-1}$ is greater than the shortest distance from v_0 to a vertex in $\Gamma(Q^r)^i$ (cf. Figure 3). While the former case does not cause violation of Σ_{br} in the presence of faults, the later may do. Hence, the rank of v_0 must be at least the rank of v_1 in V^{i-1}. Moreover, if there exist multiple fault edges at $\gamma^{-1}(v_0)$ then we take the maximum rank (Line D7). After computing the rank of vertices from where faults may occur, we adjust the rank of the rest of vertices from where faults do not occur by invoking the subroutine AdjustShortestPath (Line D9). Figure 3 illustrates how vertex rankings work.

Now, for each portion G^i, we construct a subgraph of G^i whose longest distance from each vertex in $\Gamma(P^r)^i$ to a vertex in $\Gamma(Q^r)^i$ is at most δ as follows (lines D11-D16). To this end, we begin with an empty digraph $G'^i\langle V'^i, A'^i\rangle$ and we first include shortest paths from each vertex $v \in \Gamma(P^r)^i$ to a vertex in $\Gamma(Q^r)^i$, provided $Rank(v) \leq \delta$ (lines D13-D16). Note that, adding such shortest paths does not create new paths of length greater than δ. Next, we include the remaining arcs and vertices in G'^i, so that no arcs of the form (v_0, v_1), where v_0 is on a path from $\Gamma(P^r)^i$ to $\Gamma(Q^r)^i$ are added (lines D17, D18).

Now, we transform the digraph G' back into a region graph (Line D19). Finally, we return the set $\psi_p'^r$ of edges from where Σ_{br} may not be violated even in the presence of faults, and the set ns of regions from where Σ_{br} may be violated in the presence of faults (lines D20, D21).

After adding bounded-time recovery, the algorithm Add_HardFailsafe first identifies the set rs of regions and the set rt of edges from where faults alone may violate Σ_{br} (lines B6, B7 in Figure 1). Then, it removes such regions and edges along with the deadlock regions from S^r (due to pruning some vertices and arcs in step B5) in the same fashion that we did for adding soft-failsafe fault-tolerance (Line B8). However, while removing deadlock regions, we need to consider a special case where a region $r_0 \in Q^r$ becomes a deadlock region. In this case, it is possible that all the regions along the paths that start from a region in P^r and end in r_0 become deadlock regions. Hence, we need to find another path from the region in P^r to a region in Q^r other than r_0. Hence, in this case, we remove r_0 from S^r and Q^r and start over (lines B10-B14). Finally, the algorithm ensures closure of the invariant (Line B15) and transforms the synthesized region graph $R(\mathcal{P}')$ back to a real-time program $\mathcal{P}'$ (Line B16).

Theorem 5.3. The algorithm Add_HardFalisafe is sound and complete. $\square$

Theorem 5.4. The problem of adding hard-failsafe fault-tolerance to a real-time program, where the synthesized program is required to satisfy at most one bounded response property in the presence of faults, is in PSPACE. $\square$

5.2 Automated Addition of Nonmasking Tolerance to Real-Time Programs

To derive a nonmasking f-tolerant program $\mathcal{P}'$, we ensure that if the state of $\mathcal{P}'$ is perturbed by faults in f then it recovers to a state in S within a pre-specified recovery time δ. Since a nonmasking program is not required to satisfy its safety specification in the presence of faults, to provide bounded-time recovery, it suffices to invoke the subroutine Add_BoundedRecovery for state predicates $S_p - S$ and S. Since an algorithm for adding nonmasking fault-tolerance is very simple and, in Subsection 5.3, we describe how we add bounded-time recovery from fault-span to invariant, we do not present the algorithm in a formal fashion.

5.3 Automated Addition of Masking Tolerance to Real-Time Programs

As mentioned in Section 3, in masking fault-tolerance the program is required to satisfy its safety specification in the presence of faults and if the state of a program is perturbed by faults then it recovers to its invariant within a bounded amount of time. In Subsection 5.3.1, we present our synthesis algorithm for adding soft-masking fault-tolerance to an existing real-time program. Then, in Subsection 5.3.2, we discuss the issues in addition of hard-masking fault-tolerance, where the synthesized program is required to satisfy at most one bounded response property in the presence of faults.

5.3.1 Adding Soft-Masking Fault-Tolerance

In order to synthesize a soft-masking program, we should generate a program $\mathcal{P}'$ with invariant S' and fault-span T', such that it never violates its safety specification and if a fault perturbs the state of a program to a state in T', it recovers to S' within a pre-specified recovery time δ. To this end, we extend the algorithm proposed in [6] for adding masking fault-tolerance to untimed programs, such that it provides bounded time recovery.

Now, we describe the algorithm Add_SoftMasking (cf. Figure 4) in detail. First, we construct the region graph $R(\mathcal{P})$ (Line E1). Our first estimate of a soft-masking program is a soft-failsafe program. Hence, we let our first estimate S_1^r be the region invariant of its soft-failsafe fault-tolerant program. Likewise, we estimate

```
Add_SoftMasking($\mathcal{P}\langle S_p, \psi_p \rangle$ :real-time program $f$ :transitions, $S$: state predicate, $\Sigma_{bt}$: specification, $n, \delta$: **integer**)
{
    $R(\mathcal{P})\langle S_p^r, \psi_p^r \rangle$, $S^r$, $f^r$, $\Sigma_{bt}^r$ := ConstructRegionGraph($\mathcal{P}\langle S_p, \psi_p \rangle$, $S$, $f$, $\Sigma_{bt}$);    (E1)

    Define $ms$ and $mt$ as in Add_UntimedFailsafe.    (E2)
    $S_1^r, T_1^r$ := RemoveDeadlocks($S^r - ms, \psi_p^r - mt$), $S_p^r - ms$;    (E3)

    **repeat**    (E4)
        $T_2^r, S_2^r := T_1^r, S_1^r$;    (E5)
        $\psi_{p_1}^r := \psi_p^r | S_1^r \ \cup \ \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in T_1^r \ \wedge \ (s_0, \rho_0) \notin S_1^r \ \wedge \ (s_1, \rho_1) \in T_1^r \ \wedge$
                $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \ : \ (\exists \lambda \subseteq X \ : \ \rho_1 = \rho_2[\lambda := 0])\} - mt$;    (E6)
        $T_1^r$ := ConstructFaultSpan($T_1^r - \{r \mid S_1^r$ is not reachable from $r$ using $\psi_{p_1}^r \}$, $f^r$);    (E7)
        $S_1^r$ := RemoveDeadlocks($S_1^r \wedge T_1^r, \psi_{p_1}^r$);    (E8)
        **if** ($S_1^r = \{\} \ \vee \ T_1^r = \{\}$) **then**    (E9)
            declare no soft-masking $f$-tolerant program $\mathcal{P}'$ exists; **exit**;    (E10)
    **until** ($T_1^r = T_2^r \ \wedge \ S_1^r = S_2^r$);    (E11)

    $\psi_p'^r, ns$ := Add_BoundedRecovery($R(\mathcal{P})\langle S_p^r, \psi_{p_1}^r \rangle, f^r, T_1^r - S_1^r, S_1^r, n, \delta$);    (E12)

    $rs := \{r_0 \mid \exists r_1, r_2, ...r_n : (\forall j : 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \ \wedge r_n \in ns\}$;    (E13)
    $rt := \{(r_0, r_1) \mid (r_0, r_1) \in \psi_{p_1}^r \ \wedge \ r_1 \in rs)\}$;    (E14)
    $S_1^r$ := RemoveDeadlocks($S_1^r - rs, \psi_{p_1}^r - rt$);    (E15)
    **if** ($S_1^r = \{\}$) **then** declare no soft-masking $f$-tolerant program $\mathcal{P}'$ exists; **exit**;    (E16)
    **else** $\psi_p^r$ := EnsureClosure($\psi_{p_1}^r - rt, S_1^r - rs$);    (E17)

    $S'^r, T'^r := S_1^r - rs, T_1^r - ns$;    (E18)
    $\mathcal{P}'\langle S_p, \psi_p' \rangle, S', T'$ := ConstructRealTimeProgram($R(\mathcal{P}')\langle S_p^r, \psi_p'^r \rangle, S'^r, T'^r$)    (E19)
}
ConstructFaultSpan($T^r$ : region predicate, $f^r$ : set of edges)
// *Returns* the largest subset of $T^r$ that is closed in $f^r$.
{
    **while** ($\exists r_0, r_1 : r_0 \in T^r \wedge r_1 \notin T^r \wedge (r_0, r_1) \in f^r$)
        $T^r := T^r - \{r_0\}$;
    **return** $T^r$
}
```

Figure 4: Addition of Soft-Masking Fault-Tolerance

T'^r to be T_1^r where T_1^r includes all the regions in the region space minus the regions from where safety of $R(\mathcal{P})$ may be violated (lines E2, E3). Next, we compute the set of edges $\psi_{p_1}^r$, region fault-span T_1^r, and region invariant S_1^r of $R(\mathcal{P})$ in a loop as follows (lines E4-E11):

1. In order to compute the set of edges $\psi_{p_1}^r$, we first include edges in $\psi_p^r | S_1^r$. Then we consider edges that start from a region (s_0, ρ_0), where $(s_0, \rho_0) \in T_1^r - S_1^r$, and end at a region $(s_1, \rho_1) \in T_1^r$ (by closure of fault-span), such that the time monotonicity condition is preserved, i.e., there exists ρ_2, where ρ_2 is a time-successor of ρ_0 and $\rho_1 = \rho_2[\lambda := 0]$, such that λ is any subset of the set X of clock variables of $\mathcal{P}$. Finally, we remove the transitions mt from this set (Line E6).

2. We recompute the region fault-span by first removing the regions from where S_1^r is not reachable using the edges in $\psi_{p_1}^r$. Then, we remove regions from where closure of fault-span may be violated through a fault edge, by invoking the subroutine ConstructFaultspan (Line E7).

3. Since S_1^r must be a subset of T_1^r, we recompute the region invariant by invoking the subroutine RemoveDeadlocks for $S_1^r \wedge T_1^r$ (Line E8) and jump back to step 1.

Upon the termination of the repeat-until loop, recovery from T_1^r to S_1^r is provided, but not in δ time units. Hence, we need to ensure that any computation of the soft-masking program $\mathcal{P}'\langle S_p, \psi'_p\rangle$ that starts from a state in the fault-span T', reaches its invariant S' within δ time units, even in the presence of faults. In fact, we need bounded-time recovery from each state in $T' - S'$ to a state S', which is in turn the bounded response property $(T - S) \mapsto_{\leq \delta} S$. To this end, we invoke the subroutine Add_BoundedRecovery with parameters $R(\mathcal{P})\langle S_p^r, \psi_{p1}^r\rangle, f^r, T_1^r - S_1^r, S_1^r, n$, and δ (Line E12). Since S_1^r is closed in ψ_{p1}^r, unlike adding hard-failsafe, we do not need to worry about removal of regions in S_1^r. However, if there exists a region $r_0 \in S_1^r$ that may reach a region $r_1 \in ns$ by taking faults alone, where ns is the set of regions from where recovery from T_1^r is not possible, r_0 becomes a region from where a program computation goes to the fault-span, but cannot recover to the invariant in δ time units. Hence, we remove the regions (respectively, edges), from where by taking faults alone a computation may reach a region in ns, from S_1^r (respectively, ψ_{p1}^r) (lines E13-E17). Finally, we construct the real-time program $\mathcal{P}'\langle S_p, \psi'_p\rangle$ with invariant S' out of its region graph $R(\mathcal{P}')\langle S_p^r, \psi_p'^r\rangle$ and region invariant S'^r (lines E18, E19).

Theorem 5.5. The algorithm Add_SoftMasking is sound and complete. □

Theorem 5.6. The problem of adding soft-masking fault-tolerance to a real-time program is in PSPACE. □

5.3.2 Adding Hard-Masking Fault-Tolerance with a Single Bounded Response Property

To design a hard-masking fault-tolerant program $\mathcal{P}'$ from an intolerant program $\mathcal{P}$ for the case where $\Sigma_{br} \equiv P \mapsto_{\leq \delta} Q$, we ensure that $\mathcal{P}'$ is soft-masking fault-tolerant and it maintains $P \mapsto_{\leq \delta} Q$ even in the presence of faults. Note that, since $\mathcal{P}'$ is supposed to be a soft-masking program, it must provide bounded-time recovery, which is in turn the bounded response property $(T - S) \mapsto_{\delta'} S$. In other words, $\mathcal{P}'$ must satisfy two bounded response properties simultaneously. A possible solution seems to be adding the bounded response properties one after another. Note, however, that during the addition of the first property, we may unnecessarily remove a transition that should have been kept in order to be able to add the second property. Hence, such a solutions is sound but not complete.

In this context, we note that in [31], the authors show that adding two unbounded response (leads-to) properties to an untimed program is NP-hard in the state space. While there are subtle differences between the problem considered in [31] and the problem of adding hard-masking (e.g., $P, Q \subseteq T$), based on [31], we conjecture that the time complexity of adding hard-masking fault-tolerance even with a single bounded response property is exponential in the size of the region graph.

5.4 Adding Hard Masking Fault-Tolerance with Multiple Bounded Response Properties

In this subsection, we note that if Σ_{br} consists of multiple bounded response properties then adding hard masking fault-tolerance to a real-time program is NP-hard.

Theorem 5.7. The problem of adding hard masking fault-tolerance to a real-time program where the resulting program is required to satisfy multiple bounded response properties in the presence of faults, is NP-hard in the size of region graph.

While we omit the proof of this theorem for reasons of space, we note that in [31], we have shown that the problem of adding two (unbounded) response properties to a given program (in the absence of faults) is NP-hard. The same proof can be extended to this problem, as adding hard fault-tolerance requires that bounded liveness properties are preserved in the presence of faults. For reasons of space, we refer the reader to [29] for proofs.

6 Discussion

In this section, we justify our assumptions and effect of them on our approach.

Modeling safety specification. We choose to model the untimed part of safety specifications by a set of bad transitions due to the recent results on time complexity of synthesis algorithms that deal with more general class of specifications. In [32], Kulkarni and Ebnenasir show that the problem of adding masking fault-tolerance to untimed programs, where the safety specification is specified in terms a set of bad pairs of transitions, is NP-hard. Furthermore, as mentioned in Subsection 5.4, if the safety specification consists of multiple bounded response properties, the problem of adding hard masking fault-tolerance is also NP-hard in size of region graph. Therefore, we argue that automated synthesis of both soft and hard fault-tolerant real-time programs is likely to be more successful if one focuses on problems where the untimed part safety can be represented by a set of bad transitions. Moreover, we argue that automated synthesis methods for adding hard fault-tolerance is more successful, if timing constraints of safety is represented by at most one bounded response property.

Safety specification in the absence and presence of faults. In many systems, the safety requirements in the presence of faults may be weaker than that in the absence of faults. In this case, during synthesis, one should specify the properties that should be met in the presence of faults. Since we begin with a fault-intolerant program that meets the specification in the absence of faults and no new behaviors are added in the absence of faults, the fault-tolerant program would continue to satisfy the stronger specification in the absence of faults.

Unbounded number of faults. In our work, for hard fault-tolerance we assumed that the number of fault occurrences in a computation is bounded. Note that if the number of faults are unbounded then for most interesting scenarios, the synthesis is not feasible. To illustrate this, observe that for most faults considered in practice, the occurrence of faults causes a delay in satisfaction of a bounded response property. Thus, if unbounded number of faults occur then hard fault-tolerance cannot be satisfied unless we ensure that the program does not reach states where faults cannot occur.

State space explosion problem . Region graph is usually not the most efficient finite representation of a real-time program. By contrast, zone automata [33] is considered as a more efficient model used in model checking techniques. In this paper, since our goal was to investigate the possibility of synthesizing fault-tolerant real-time programs and to evaluate the classes of complexity of such algorithms, we focused on detailed region graphs. However, an interesting improvement step is modifying the algorithms presented in Section 5, so that manipulate a zone automaton rather than a region graph during synthesis.

7 Conclusion and Future Work

In this paper, we focused on the problem of automatic addition of fault-tolerance to real-time programs. We considered three levels of fault-tolerance, failsafe, nonmasking, and masking. For failsafe and masking, we proposed two cases, soft and hard, based on satisfaction of timing constraints in the presence of faults. In our approach, we begin with an existing program rather than specification and, hence, the previous efforts made for synthesizing the input program are reused.

We first introduced a generic framework to formally define the notions of faults and fault-tolerance in the context of real-time programs. Then, we presented sound and complete algorithms for transforming fault-intolerant real-time programs into soft-failsafe, nonmasking, and soft-masking fault-tolerant programs. We also proposed a sound and complete algorithm that synthesizes hard-failsafe fault-tolerant real-time programs, where the fault-tolerant program is required to satisfy at most one bounded response property in the presence of faults. The complexity of our algorithms are in polynomial time in the size region graphs. We also showed that the problem of adding hard masking fault-tolerance to real-time programs, where the fault-tolerant program is required to satisfy multiple bounded response properties in the presence of faults, is NP-hard. Thus, this work characterizes classes of problems where adding fault-tolerance to real-time programs is expected to be feasible and where the complexity is too high.

Since the complexity of the aforementioned algorithms is comparable to that of existing model checking techniques, we believe that the proposed algorithms can be used in tools for synthesizing fault-tolerant real-time programs in practice. More specifically, as future work, we plan to extend our tool FTSyn [2] (which is currently capable to synthesize fault-tolerant untimed programs), so that it synthesizes fault-tolerant real-time programs as well.

References

[1] P. M. Alvarez and D. Mossé. A responsiveness approach for scheduling fault recovery in real-time systems. In *IEEE Real Time Technology and Applications Symposium*, pages 4–13, 1999.

[2] F. Liberato, R. G. Melhem, and D. Mossé. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transations on Computers*, 49(9):906–914, 2000.

[3] P. M. Alvarez, H. Aydin, D. Mossé, and R. G. Melhem. Scheduling optional computations in fault-tolerant real-time systems. In *Real-Time Computing Systems and Applications (RTCSA)*, 2000.

[4] H. Aydin, R. G. Melhem, and D. Mossé. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 289–296, 2000.

[5] D. Mossé, R. G. Melhem, and S. Ghosh. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Transactions on Software Engineering*, 29(8):752–767, 2003.

[6] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 1926 of *LNCS*, pages 82–93, 2000.

[2]For more information, visit `http://www.cse.msu.edu/˜ebnenasi/research/tools/ftsyn.htm.`

[7] S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *20th Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.

[8] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. In *International Conference on Distributed Computing Systems(ICDCS)*, 2003.

[9] S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.

[10] S. S. Kulkarni and A. Ebnenasir. Adding fault-tolerance using pre-synthesized components. In *European Dependable Computing Conference (EDCC)*, pages 72–90, 2005.

[11] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems*, 26(1):125–185, January 2004.

[12] H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete event systems. In *30th International Conference on Decision and Control*, pages 1527–1528, Brighton, UK, 1991.

[13] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. *STACS'95 (12th Annual Symposium on Theoretical Aspects of Computer Science)*, pages 229–242, 1995.

[14] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. *IFAC Symposium on System Structure and Control*, pages 469–474, 1998.

[15] E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569 of *LNCS*, pages 19–30. Springer, March 1999.

[16] D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *STACS*, pages 571–582, 2002.

[17] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification*, pages 180–192, 2003.

[18] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *14th Internation Conference on Concurrency Theory*, volume 2761 of *LNCS*. Springer-Verlag, 2003.

[19] M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science*, pages 167–176, 2002.

[20] S. Tripakis. Fault diagnosis for timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 205–224, 2002.

[21] P. Bouyer, F. Chevalier, and D. D'Souza. Fault diagnosis using timed automata. In *Foundations of Software Science and Computation Structure*, pages 219–233, 2005.

[22] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[23] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal on Software Tools for Technology Transfer*, 1(1-2):86–109, 1997.

[24] T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.

[25] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[26] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 7 October 1985.

[27] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[28] S. S. Kulkarni. *Component-based design of fault-tolerance*. PhD thesis, Ohio State University, 1999.

[29] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Automatic addition of fault-tolerance to real-time programs. Available at `http://www.cse.msu.edu/~borzoo/proofs.pdf`.

[30] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Computer-Aided Verificaion*, LNCS 575:399–409, 1991.

[31] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *9th International Conference on Principles of Distributed Systems*, 2005. To appear.

[32] S. S. Kulkarni and A. Ebnenasir. The effect of the specification model on the complexity of adding masking fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(4):348–355, October-December 2005.

[33] R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *3rd International Conference on Concurrency Theory*, pages 340–354, 1992.